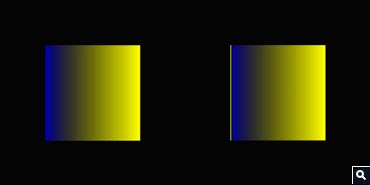
**The OpenGL Pipeline Newsletter - Volume 003**

[Table of Contents](https://www.opengl.org/pipeline/vol003)  
Previous article: [ARB Next Gen TSG Update](https://www.opengl.org/pipeline/article/vol003_5)  
Next article: [OpenGL and Windows Vista™](https://www.opengl.org/pipeline/article/vol003_7)

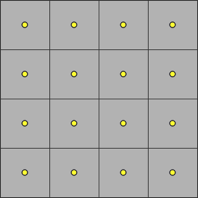
**GLSL: Center or Centroid?  (Or When Shaders Attack!)**

[](https://www.opengl.org/img/uploads/pipeline/vol003/correct-incorrect-800.jpg)  
Figure 1 - Correct (on left) versus Incorrect (on right). Note the yellow on the left edge of the “Incorrect” picture. Even though myMixer varies between 0.0-1.0, somehow myMixer is outside that range on the “Incorrect” picture. [View Closeup](https://www.opengl.org/img/uploads/pipeline/vol003/correct-incorrect-800.jpg)

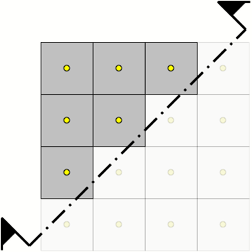
Let's take a look at a simple fragment shader but with a simple non-linear twist:

varying float myMixer;  
  
// Interpolate color between blue and yellow.  
// Let's do a sqrt for a funkier effect.  
void main( void )  
{  
    const vec3 blue   = vec3( 0.0, 0.0, 1.0 );  
    const vec3 yellow = vec3( 1.0, 1.0, 0.0 );  
    float a = sqrt( myMixer ); // undefined when myMixer < 0.0  
    vec3 color = mix( blue, yellow, a ); // nonlerp  
    gl\_FragColor = vec4( color, 1.0 );  
}

How did the yellow stripe on the “Incorrect” picture get there?  To best understand what went wrong, let's first examine the case where it will (almost) always be “Correct.”  That case is single sample rendering.

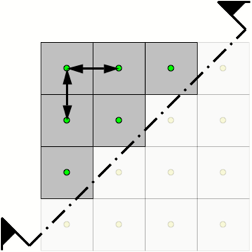
  
Figure 2 - Squares with yellow dots.

This is classic single sample rasterization.  Grey squares represent the pixel square (or a box filter around the pixel center).  Yellow dots are the pixel centers at half-integer window coordinate values.

  
Figure 3 - Squares with yellow dots, halved.

The section line represents a half-space of a primitive.  Above and to the left the associated data myMixer is positive.  Below and to the right it is negative.

In classic single sample rasterization an in/out/on classification at each pixel center will produce a fragment for pixel centers that are “in” the primitive.  The six fragments in this example that must be produced are in the upper left.  Those pixels that are “out” are dimmed, and will not have fragments generated.

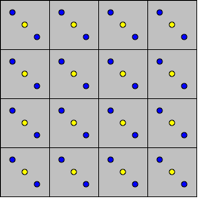
  
Figure 4 - Squares with green dots halved with arrows.

Green dots show where shading will take place for each of the six fragments.  The associated data myMixer is evaluated at each pixel center.  Note that each of the green dots are above and to the left of the half-space, therefore they are all positive.  All of the associated data is interpolated.

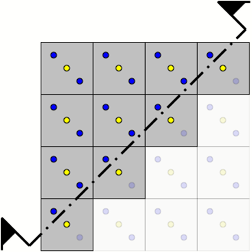
While our simple shader uses no derivatives (explicit or implied, such as with mipmapped or anisotropic texture fetches), the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow).  In the interior of the primitive they are quite well defined and regular.

Bottom line: with single sampling, fragments are *only* generated if the pixel center is classified “in,” fragment data is evaluated at the pixel center, interpolation only happens within the primitive, and shading only takes place within the primitive.  All is good and “Correct.”  (Almost always.  For now we'll ignore inaccuracies in some of the derivatives on pixels along the edge of the half-space.)

So, all is (almost) well with single sample rasterization.  What can go wrong with multi-sample rasterization?

  
Figure 5 - Squares with yellow and blue dots.

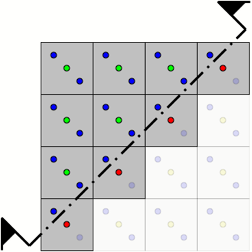
This is classic multi-sample rasterization.  Grey squares represent the pixel square (or a box filter around the pixel center).  Yellow dots are the pixel centers at half-integer window coordinate values.  Blue dots are sample locations.  In this example, I'm showing a simple rotated two sample implementation.  Everything generalizes to more samples.

  
Figure 6 - Squares with yellow and blue dots halved.

The section line again represents a half-space of a primitive.  Above and to the left the associated data myMixer is positive.  Below and to the right it is negative.

In multi-sample rasterization an in/out/on classification at each sample will produce a fragment if *any* sample associated with a pixel is “in” the primitive.

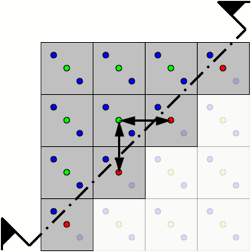
The ten fragments in this example that must be produced are in the upper left.  (Note the four additional fragments generated along the half-space.  One sample is “in” even though the center is “out.”)  Those pixels that are “out” are dimmed.

  
Figure 7 - Squares with yellow blue green and red dots halved.

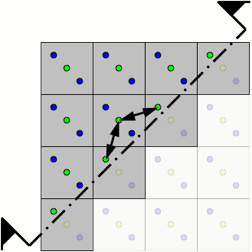
What if we evaluate at pixel centers?

Green dots *and* red dots show where shading will take place for each of the ten fragments.  The associated data myMixer is evaluated at each pixel center.  Note that each of the green dots are above and to the left of the half-space, therefore they are all positive.  But also note that each of the red dots are below and to the right of the half-space, therefore they are negative.  The green dots are where the associated data is interpolated, red dots are where they are extrapolated.

In the example shader, sqrt(myMixer) is undefined if myMixer is negative.  Even though the values written by a vertex shader might be in the range 0.0-1.0, due to the extrapolation that might happen, myMixer can be outside the range 0.0-1.0.  When myMixer is negative the result of the fragment shader is undefined!

  
Figure 8 - Squares with yellow blue green and reds dots halved with arrows.

We're still considering the case of evaluation at pixel centers.  While our simple shader uses no derivatives, explicit or implied, the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow).  In the interior of the primitive they are quite well defined because all of the evaluation is at the regular pixel centers.

  
Figure 9 - Squares with yellow blue and green dots halved with arrows.

What if we evaluate other than at pixel centers?

Green dots show where shading will take place for each of the ten fragments.  The associated data myMixer is evaluated at each pixel “centroid.”

The pixel centroid is the center of gravity of the intersection of a pixel square and the interior of the primitive.  For a fully covered pixel this is exactly the pixel center.  For a partially covered pixel this is often a location *other* than the pixel center.

OpenGL allows implementers to choose the ideal centroid, or any location that is inside the intersection of the pixel square and the primitive, such as a sample point or a pixel center.

In this example, if the center is “in,” the associated data is evaluated at the center.  If the center is “out,” the associated data is evaluated at the sample location that is “in.”  Note that for the four pixels along the half-space the associated data is evaluated at the sample.

Also note that each of the green dots are above and to the left of the half-space.  Therefore, they are all positive: always interpolated, never extrapolated!

So why not always evaluate at centroid?  In general, it is more expensive than evaluating at center.  But that's not the most important factor.

While our simple shader uses no derivatives, the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow).  Note that the spacing between evaluations is not regular.  They also do not hold y constant for dFdx, or hold x constant for dFdy.  *Derivatives are less accurate when evaluated at centroid!*

Because this is a tradeoff, OpenGL Shading Language Version 1.20 gives the shader writer the choice of when to make the tradeoff with a new qualifier, centroid.

#version 120  
  
**centroid** varying float myMixer;  
  
// Interpolate color between blue and yellow.  
// Let's do a sqrt for a funkier effect.  
void main( void )  
{  
    const vec3 blue   = vec3( 0.0, 0.0, 1.0 );  
    const vec3 yellow = vec3( 1.0, 1.0, 0.0 );  
    float a = sqrt( myMixer ); // undefined when myMixer < 0.0  
    vec3 color = mix( blue, yellow, a ); // nonlerp  
    gl\_FragColor = vec4( color, 1.0 );  
}

When should you consider using centroid?

1. When using an extrapolated value could lead to undefined results.  Pay particular attention to the built-in functions that say “results are undefined if!”
2. When using an extrapolated value with a highly non-linear or discontinuous function. This includes for example specular calculations, particularly when the exponent is large, and step functions.

When should you *not* consider using centroid?

1. When you need accurate derivatives (explicit or implied, such as with mipmapped or anisotropic texture fetches).  The shading language specification considers derivatives derived from centroid varings to be so fraught with inaccuracy that it was resolved they are simply undefined.  In such a case, strongly consider  at least adding:      centroid varying float myMixer; // beware of derivative!  
        varying float myCenterMixer; // derivative okay
2. With tessellated meshes where most of the quad or triangle boundaries are interior and well defined anyway.  The easiest way to think about this case is if you have a triangle strip of 100 triangles, and only the first and last triangle might result in extrapolations, centroid will make those two triangles interpolate but at the tradeoff of making the other 98 triangles a little less regular and accurate.
3. If you know there might be artifacts from undefined, non-linear, or discontinuous functions, but the resulting artifacts are nearly invisible.  If the shader is *not* attacking (much), don't fix it!

**Bill Licea-Kane, AMD**  
**Shading Language Technical SubGroup Chair**